Reconciling Synthesis and Decomposition:
A Composite Approach to Capability Identification

Ramya Ravichandar and James D. Arthur
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060
Email: {ramyar, arthur}@vt.edu

Robert P. Broadwater
Electrical and Computer Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24060
Email: {dew}@vt.edu

Abstract—**Stakeholders' expectations and technology constantly evolve during the lengthy development cycles of a large-scale computer based system. Consequently, the traditional approach of baselining requirements results in an unsatisfactory system because it is ill-equipped to accommodate such change. In contrast, systems constructed on the basis of Capabilities are more change-tolerant; Capabilities are functional abstractions that are neither as amorphous as user needs nor as rigid as system requirements. Alternatively, Capabilities are aggregates that capture desired functionality from the users' needs, and are designed to exhibit desirable software engineering characteristics of high cohesion, low coupling and optimum abstraction levels. To formulate these functional abstractions we develop and investigate two algorithms for Capability identification: Synthesis and Decomposition. The synthesis algorithm aggregates detailed rudimentary elements of the system to form Capabilities. In contrast, the decomposition algorithm determines Capabilities by recursively partitioning the overall mission of the system into more detailed entities. Empirical analysis on a small computer based library system reveals that neither approach is sufficient by itself. However, a composite algorithm based on a complementary approach reconciling the two polar perspectives results in a more feasible set of Capabilities. In particular, the composite algorithm formulates Capabilities using the cohesion and coupling measures as defined by the decomposition algorithm and the abstraction level as determined by the synthesis algorithm.**

I. INTRODUCTION

The property of change-tolerance is of paramount importance in complex emergent systems. These computer based systems are of large magnitude, have lengthy development cycles and are envisioned to be utilized for an extended lifetime. In addition, their inherent complexity results in emergent behavior [1] that is often unexpected. For example, the introduction of a new functionality in the system may result in unanticipated interactions with other existing components that can be detrimental to the overall system functionality. Moreover, in order to function satisfactorily complex emergent systems must accommodate the effect of dynamic factors such as varying expectations of the stakeholders, changing user needs, technology advancements, scheduling constraints and market demands, during their lengthy development periods. We conjecture that these changes can be accommodated with minimum impact, if systems are architected using aggregates that are embedded with change-tolerant characteristics. We term such aggregates as *Capabilities*. Capabilities are functional abstractions that exhibit high cohesion, low coupling and balanced abstraction levels. The property of high cohesion helps localize the impact of change to within a Capability. Also, the ripple effect of change is less likely to propagate beyond the affected Capability because of its reduced coupling with neighboring Capabilities. An optimum level of abstraction assists in the understanding of the functionality in terms of its most relevant details [2]. In addition, we observe that the abstraction level is related to the size of a Capability; the higher the abstraction level, the greater is the size of a Capability [3]. From a software engineering perspective, abstractions with a smaller size are more desirable for implementation. Therefore, we need to design an algorithm based on the three characteristics of cohesion, coupling and abstraction, that in some sense, "optimizes" the identification of Capabilities. Specifically, we use a top-down and a bottom-up approach as the basis of the algorithms for formulating Capabilities. This is because our cognitive ability to examine a problem from both a top-down and a bottom-up perspective facilitates the application of widely diverse solution approaches. This phenomenon is evident in the field of software engineering where development strategies such as top-down design, bottom-up testing, top-down integration and others that incorporate a top-down or a bottom-up perspective are utilized in the different stages of system development. In particular, for Capability identification we focus on needs analysis, a phase prior to requirements specification, because Capabilities are formulated from user needs. At this point we consider only the functional aspects of the system. Following convention, we develop two algorithms for Capability identification that are based on the top-down and bottom-up approaches:

- *Synthesis:* This is an algorithm based on the bottom-up approach. The system is understood in terms of its most detailed elements, which are then systematically aggregated to form abstractions of higher levels.
- *Decomposition:* This is an algorithm based on the top-down approach. The system is visualized in terms of its highest level mission, which is then systematically decomposed into abstractions that are more detailed.

In either approach the objective is to identify functional abstractions that are maximally cohesive and minimally coupled

as Capabilities. We assessed the efficacy of the synthesis and decomposition algorithms by executing them on a real-world computer based library system. Our empirical analysis reveals that neither approach is sufficient by itself to determine the best set of Capabilities. More specifically, the cohesion measure based on the synthesis approach is inordinately subjective. Additionally, the synthesis strategy provides little information to assist coupling measurements. However, this approach identifies aggregates of reduced sizes as Capabilities. In contrast, the decomposition approach expedites the measurement of cohesion and coupling but results in Capabilities that are of increased sizes. In other words, these Capabilities are defined at very high levels of abstraction. Therefore, we construct a composite algorithm to establish an equilibrium between the two polar approaches. This algorithm is based on a complementary approach that incorporates elements of cohesion and coupling from the decomposition strategy, and models abstraction from the synthesis perspective.

The remainder of the paper is organized as follows: Section II discusses related work and outlines the overall process of engineering Capabilities. In Section III and Section IV, we compare and contrast the three primary elements that determine a Capability — cohesion, coupling, and abstraction level — from the synthesis and decomposition approaches, respectively. In Section V, we describe our composite algorithm that combines the two approaches. Our conclusions are presented in Section VI.

II. BACKGROUND

A system operating in the real world is subject to dynamic factors of change. These factors necessitate system evolution, the process of constantly adapting to various influences in order to function satisfactorily [4]. Software development processes that are ill-equipped to accommodate change are primarily afflicted with requirements volatility [5]. This phenomenon is known to increase the defect density and affect project performance resulting in schedule and cost overruns [6] [7]. Traditional Requirements Engineering (RE) strives to manage volatility by baselining requirements. However, the dynamics of user needs and technology advancements during the extended development periods of complex emergent systems discourage fixed requirements. More recently, techniques such as the Performance based specifications [8] [9] and Capability Based Acquisition (CBA) [10] are being utilized to mitigate change in large-scale systems. Performance based specifications are requirements describing the outcome expected of a system from a high-level perspective. The less-detailed nature of these specifications provides latitude for incorporating appropriate design techniques and new technologies. Similarly, CBA is expected to accommodate change and produce systems with relevant capability and current technology. It does so by delaying requirement specifications in the software development cycle, and by allowing time for a promising technology to mature so that it can be integrated into the software system. However, the Performance based specification and the CBA approaches lack a scientific procedure

for deriving system specifications from an initial set of user needs. Moreover, they neglect to define the level of abstraction at which a specification or Capability is to be described. Thus, these approaches propose solutions that are neither definitive, comprehensive nor mature enough to accommodate change and benefit the development process for complex emergent systems.

Our approach, the Capabilities Engineering (CE) process, architects change-tolerant systems on the basis of optimal sets of Capabilities. In fact, Rowe and Leany suggest that it is beneficial to address the issues of evolution when modeling the system architecture [11]. Therefore, we design Capabilities to incorporate evolutionary-friendly characteristics such as high cohesion, minimal coupling, and pragmatic levels of functional abstraction. Figure 1 illustrates the two major phases of the CE process. Phase I identifies sets of Capabilities based on the
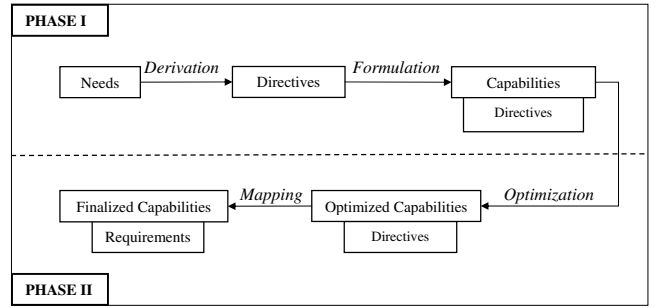

Fig. 1. *Capabilities Engineering Process*

values of cohesion, coupling and abstraction levels. Techniques of modularization suggest that high cohesion and low coupling are typical of stable units [12] [13]. Stability implies resistance to change; in the context of CE, we interpret stability as a property that accommodates change with minimum ripple effect. Ripple effect is the phenomenon of propagation of change from the affected source to its dependent constituents. Specifically, dependency links between aggregates behave as change propagation paths. The higher the number of links, the greater is the likelihood of ripple effect. Because coupling is a measure of interdependence between units [14] we choose coupling as one indicator of stability of an aggregate. In contrast, cohesion — the other characteristic of a stable structure — depicts the "togetherness" of elements within an aggregate. A unit is said to be highly cohesive if each of its elements is directed towards achieving a single objective. As a general observation as the cohesion of a unit increases, the coupling between the units decreases. However, this correlation is only approximate, and thereby, cannot be used to estimate the values of cohesion and coupling [13]. Therefore, we develop specific metrics to compute these values for potential Capabilities.

Phase II, a part of our ongoing research, further optimizes these initial sets of Capabilities to accommodate schedule constraints and technology advancements. In this paper, we focus on identifying Capabilities as outlined by Phase I.

In the following sections, we discuss the synthesis and the decomposition algorithms for computing Capabilities. We then

explain the necessity for a composite algorithm that includes elements of cohesion, coupling, and abstraction from both these approaches.

III. Synthesis

The objective of the synthesis algorithm is to formulate Capabilities — functional abstractions with high cohesion and low coupling — from user needs that are obtained during the process of elicitation [15]. Needs are affiliated with the problem domain and requirements are associated with the solution domain. Capabilities are computed after the analysis of user needs but prior to requirements specification. We envision that by doing so Capabilities can bridge the chasm between the problem and the solution space, also described as the complexity gap [16]. It is recognized that this gap is responsible for information loss, misconstrued needs, and other detrimental effects that plague system development [17] [18]. The synthesis algorithm is based on a bottom-up approach, and hence, envisions a system in terms of its details. In particular, we consider system details that are defined at low levels of abstraction and are stated from a user's perspective. We term these details as *directives*. More specifically, a directive is a system specification that is described using the terminology of the problem domain. In contrast, a requirement is a system specification stated in the technical language of the solution domain. However, both a directive and a requirement share the commonality of being defined at a low level of abstraction.

Directives are a natural derivative of user needs. We use the directives as input to the synthesis algorithm for formulating Capabilities because they serve three main purposes. Firstly, directives strive to alleviate loss of domain knowledge, which has been identified as an important problem in RE [17]. They do so by describing system functionality in terms of the problem domain. This assists in capturing domain information. Secondly, directives are utilized to compute the cohesion and coupling values of potential Capabilities. Recall that optimal sets of Capabilities are to be determined from different functional abstractions. Capabilities are essentially system functionalities, and hence, are associated with one or more directives. Therefore, the cohesion and coupling measures of Capabilities are determined using directives. Lastly, directives facilitate the mapping to system requirements. Note that Capabilities only provide a high-level architecture based on system functionalities, and therefore, requirement specifications are still necessary to direct system development. Thus, directives are easily mapped to requirements because both entities are defined at similar levels of abstraction.

A. Algorithm

The synthesis algorithm aims to identify abstractions with maximum cohesion and minimum coupling, as Capabilities. In particular, it strives to maximize functional cohesion, the most desirable cohesion among all other types of cohesion (coincidental, logical, temporal, procedural, communicational, and sequential) [19]. This objective of the synthesis algorithm is illustrated in Figure 2. If every element of a unit is essential
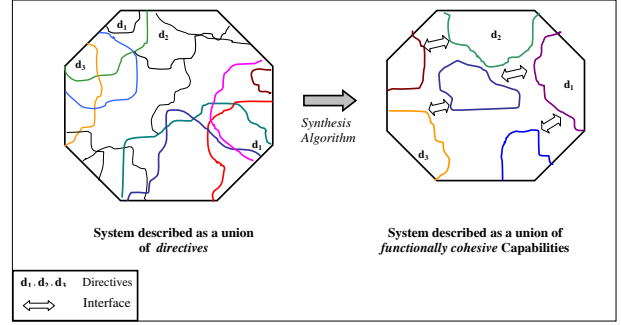

Fig. 2. *Objective of Synthesis Algorithm*

to the performance of a single function, then that unit is said to exhibit high functional cohesion [13]. Therefore, the first step of the algorithm enumerates functions that possess high functional cohesion. More specifically, we examine the significance of each directive in accomplishing various system functions. We use these significance values to compute the cohesion of a function in terms of all its participating directives. However, it is possible that the same function is described at different levels of abstraction. We represent the functions using Venn diagrams to visually understand and resolve the discrepancies in the abstraction levels. The algorithm is explained in detail next.

Let $d_1, d_2, \ldots, d_n$, $n \in \mathbb{N}$, denoting directives derived from user needs be the input to the synthesis algorithm. For each d_i perform the following steps to determine the Capabilities of a system:

1) *Identify all possible functions to which directive d_i contributes.* The relevance of a directive in accomplishing a function is estimated using the impact categories shown in Table I. This classification is intended to assess the impact of risks on a project [20]. The failure to implement a directive is also a risk, and thereby, we use this classification to determine the significance of a directive in implementing a system functionality. We assign relevance values based on the perceived significance of each impact category; these values are normalized to the [0,1] scale.

TABLE I
Relevance Values

Impact	Description	Relevance
Catastrophic	Task failure	1.00
Critical	Task success questionable	0.70
Marginal	Reduction in performance	0.30
Negligible	Non-operational impact	0.10

Formally, we enumerate the list of functions f_{im}, $m < n$, that d_i is associated with, as $InitialSet_i = \{f_{i1}, f_{i2}, \ldots, f_{im}\}$. For example, let d_1 help achieve functions $f_{1j}, j = 1, \ldots, 7$. A Venn diagram representation indicating the different abstraction levels of the functions of $InitialSet_1$ is shown in Figure 3. Late, we use

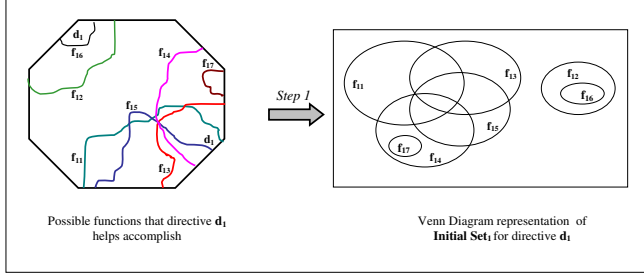the relevance values of directives later to compute the cohesion of potential Capabilities.

Fig. 3. *Example Initial Set for directive d_1*

2) *Expected system functionalities deduced from user needs can be stated at different levels of abstraction.* Consequently, certain functions constituting $InitialSet_i$ may be inclusive of other functions in the same $InitialSet_i$. For example, in Figure 3, f_{12} is inclusive of f_{16}. We avoid considering functional abstractions that are partially or completely redundant as potential Capabilities by constructing $Subset_i \subseteq InitialSet_i$ where $Subset_i = \{f_{ix}|f_{ix} \supseteq f_{iy}, \forall f_{iy} \in InitialSet_i; 1 \leqslant x, y \leqslant m\}$. Note that the functions in $Subset_i$ are not encompassed by any other function in $InitialSet_i$. This implies that $Subset_i$ consists of functions defined at the *highest* level of abstraction among all other functions in $InitialSet_i$. Thus, as shown in Figure 4 for d_1, $Subset_1 = \{f_{1j}\}, j = 1, \ldots, 5$.
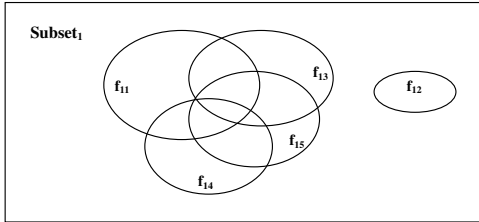

Fig. 4. *Example Subset for directive d_1*

3) *Although the aggregates in $Subset_i$ are not subset to any other aggregate, they can share common functionalities, which is an indicator of coupling.* Recall that a Capability is a self-contained functional abstraction that is minimally coupled with other Capabilities. We strive to minimize the coupling between abstractions by reducing their dependencies. Specifically, in the synthesis algorithm we use the abstraction level as an instructive factor in constructing minimally coupled aggregates. The technique of abstraction allows us to contain the dependencies within the boundaries of a higher abstraction. In particular, we identify aggregates that exhibit overlapping functionalities and aggregate them to form more decoupled abstractions. Hence, we create *aggregate subsets* $AG_{ij}(1 \leqslant i \leqslant n; 1 \leqslant j \leqslant m)$

from $Subset_i$ to contain aggregates with commonalities. Specifically,
$$AG_{ij} = \{f_{ix}, f_{iy}|f_{ix} \cap f_{iy} \neq \emptyset; 1 \leqslant x, y \leqslant m\}$$
such that $Subset_i = \cup_x f_{ix}, \forall f_{ix} \in AG_{ij}$;
We then abstract the entities of AG_{ij} to form higher level aggregates such that $AG_{ij} = \{F_{ij}\}$ where $F_{ij} = \{f_{ix} \cup f_{iy} \cup \ldots \cup f_{iz}\}; 1 \leqslant x, y, \ldots, z \leqslant m$. F_{ij} encompasses all aggregates in AG_{ij}. We term F_{ij} as **core functions**. Hence, we utilize core functions to derive and represent the functionality of system aggregates at a higher level of abstraction. For example, for directive d_1, in Figure 5, $AG_{11} = \{F_{11}\}$ where $F_{11} = \{\cup_j f_{1j}\}, j = 1, 3, 4, 5$ and $AG_{12} = \{F_{12}\}$ where $F_{12} = \{f_{12}\}$.
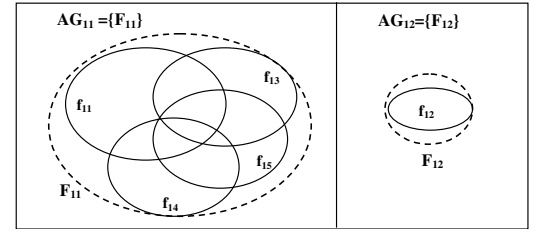

Fig. 5. *Example Aggregate Subsets for directive d_1*

4) *Let the core functions, F_{ij}, of all the aggregated subsets AG_{ij} related to directive d_i constitute the ith **Core Function Set**, CFS_i, such that $CFS_i = \{F_{i1}, F_{i2}, \ldots, F_{ij}\}; 1 \leqslant i \leqslant n; 1 \leqslant j \leqslant m$. Hence, CFS_i comprises core functions that are functional abstractions initially defined at a more detailed level. These functional abstractions are potential Capabilities. Thus, as shown in Figure 6, $CFS_1 = \{F_{11}, F_{12}\}$.*
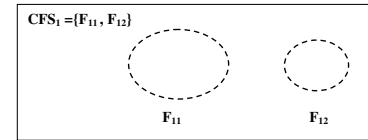

Fig. 6. *Example CFS for directive d_1*

Thus, in this manner, the synthesis algorithm defines *a Core Function Set (CFS) for each directive in the system*. Specifically, each directive d_i has an associated CFS_i. The elements of a CFS are core functions, which are aggregates derived from a systematic process of synthesizing directives.

Recall that Capabilities are functional abstractions that exhibit high cohesion and low coupling. Therefore, we now measure the cohesion and coupling values and examine the abstraction level of each core function in order to determine the set of Capabilities.

- *Cohesion*: For each directive the synthesis algorithm generates a CFS comprising core functions. The cohesion of a core function is computed as an average of the relevance values of each participating directive in achieving that

function. This implies that the list of directives associated with each core function in every CFS be enumerated; this necessitates substantial time and effort. Also, note that the core functions associated with different directives may be defined at various abstraction levels. Consequently, core functions may be subsets of one another resulting in redundant computations of relevance values. Furthermore, in our empirical analysis we observe that although the calculation of the average cohesion value is direct, the process of eliciting relevance values for each core function is highly cumbersome and notably subjective. These factors require us to explore alternate approaches for determining the cohesion of potential Capabilities.

- *Coupling*: Units are said to be coupled if changes in a source unit affect one or more dependent entities. The only information available for computing the coupling between the elements of CFSs in the synthesis algorithm is the set of common directives shared by the core functions. Experimental results show that determining coupling values merely based this number is unrepresentative of the actual implementation. Furthermore, the synthesis approach fails to provide information about the strength of dependency between functions. Hence, we conclude that the synthesis algorithm is ill-equipped to facilitate the computation of coupling between potential Capabilities.

- *Abstraction Level*: We know that each directive has an associated CFS whose elements are core functions. Empirical analysis reveals that at the abstraction level computed by the synthesis algorithm the core functions of a particular CFS do not share commonalities with other functions. However, any reduction in the abstraction level results in common intersections between aggregates. This is explained by the design of the synthesis algorithm, which terminates once a functional aggregate is established, as illustrated by the example of directive d_1. The synthesis algorithm indicates that the abstraction level of a core function is perhaps determined by examining its links with other core functions. Therefore, one needs to consider the abstraction level, and the links between aggregates when formulating Capabilities.

The synthesis algorithm attempts to identify Capabilities from the detailed directives of complex emergent systems. Given the large magnitude of these systems, considerable effort is required to establish the CFSs for 100s of directives. We note that, although the synthesis algorithm does provide insights regarding an ideal abstraction level of Capability, it is infeasible to automate the computation of cohesion and coupling measures. Therefore, it seems impractical that the synthesis algorithm be utilized for identifying Capabilities. This mandates that we design a more objective algorithm that is far less dependent on user input. Hence, we examine an alternative solution — a decomposition algorithm based on the top-down approach — in the following section.

IV. DECOMPOSITION

The decomposition algorithm utilizes a graph-based representation of user needs, *viz.* a Function Decomposition (FD) graph, to formulate Capabilities. An FD graph represents functional abstractions of the system obtained by the systematic decomposition of user needs. A need at the highest level of abstraction is the mission of the system and is represented by the root. We use the top-down philosophy to decompose the mission into functions at various levels of abstraction. We claim that a decomposition of needs is equivalent to a decomposition of functions because a need essentially represents some functionality of the system. Formally, we define an FD graph $G = (V, E)$ as an acyclic directed graph where V is the vertex set and E is the edge set. V represents the system functionality: leaves represent directives, the root symbolizes the mission, and internal nodes indicate system functions at various abstraction levels. Similarly, the edge set E comprises edges that depict decomposition, intersection or refinement relationship between nodes. These edges are illustrated in Figure 7. An edge between a parent and its child node represents functional decomposition and implies that the functionality of the child is a proper subset of the parents functionality. Only internal (non-leaf) nodes with an outdegree of at least two can have valid decomposition edges with their children. The refinement relation is used when there is a need to express a node's functionality with more clarity, say, by furnishing additional details. A node with an outdegree of one symbolizes this type of relationship with its child node. To indicate the commonalities between functions defined at the same level of abstraction the intersection edge is used. Hence, a child node with an indegree greater than one represents a functionality common to all its parent nodes. The FD graph utilizes these definitions to provide a structured top-down representation of system functionality, and thereby, facilitates the decomposition algorithm to formulate Capabilities in terms of their cohesion, coupling, and abstraction values. We discuss the mechanics of the algorithm next.
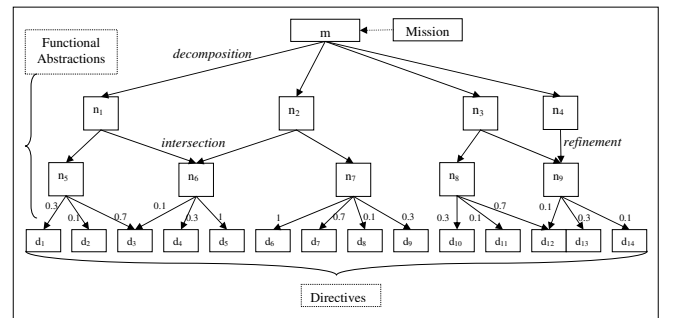

Fig. 7. *Example FD Graph $G = (V, E)$*

A. Algorithm

The input to the decomposition algorithm is an FD graph, $G = (V, E)$ that represents the functionality of the system to be developed. We first determine the set of all valid

combinations of internal nodes that can be considered as potential Capabilities. These combinations are termed *slices*. Then we compute the cohesion and coupling measures for each slice and examine the levels of abstraction to establish the finalized set of Capabilities.

We define slice S as a subset of V where the following constraints are satisfied:

1) *Complete Coverage of Directives:* We know that a Capability is associated with a set of directives, which is finally mapped to system requirement specifications (see Figure 1). Consequently, a set of Capabilities of the system has to encompass all the directives derived from user needs. The leaves of the FD graph constitute the set of all directives in a system. We ensure that each directive is accounted for by some Capability, by enforcing the constraint of complete coverage given by $\bigcup_{i=1}^{m} D_i = L$, where
   - D_i denotes the set of leaves associated with the ith node of slice S
   - $L = \{u \in V | outdegree(u) = 0\}$ denotes the set of all leaves of G
   - $m = |S|$

2) *Unique Membership for Directives:* In the context of directives, by ensuring that each directive is uniquely associated with exactly one Capability, we avoid implementing redundant functionality. Otherwise, the purpose of using slices to determine Capabilities as unique functional abstractions is defeated. We ensure the unique membership of directives by the constraint $\bigcap_{i=1}^{m} D_i = \{\phi\}$.

3) *System Mission is not a Capability:* The root is the high level mission of the system and cannot be considered as a Capability. The cardinality of a slice containing the root can only be one. This is because, including other nodes with the root in the same slice violates the second constraint. Hence, $\forall u \in S, indegree(u) \neq 0$.

4) *Directive is not a Capability:* A leaf represents a directive, which is a system characteristic. A slice that includes a leaf fails to define the system in terms of its functionality and focuses on describing low level details. Hence, $\forall u \in S, outdegree(u) \neq 0$.

The cohesion and coupling values for each slice is computed using the measures described next. We also discuss the average abstraction level of nodes that possess high cohesion and low coupling values.

- *Cohesion*: As in the synthesis algorithm, the cohesion of a node in a slice is computed as an average of the relevance values of the participating directive. The relevance values are assigned based on the values listed in Table I. However, we make a distinction between the parent and ancestor nodes of a directive. In order to reduce the need for user input, we elicit the relevance value of a directive only with respect to its parent node, whose cohesion is the arithmetic mean of the relevance values

of its directives. Figure 7 illustrates relevance values of directives to their parents. However, the cohesion of an ancestor is computed as a weighted average of the size (number of associated directives) and cohesion of its non-leaf children. Specifically, the cohesion measure of an internal node n with $t > 1$ non-leaf children is:

$$Ch(n) = \frac{\sum_{i=1}^{t} (size(v_i).Ch(v_i))}{\sum_{i=1}^{t} size(v_i)}$$

such that $(n, v_i) \in E$ and,

$$size(n) = \begin{cases} \sum_{i=1}^{t} size(v_i) & (n, v_i) \in E; outdegree(v_i) > 0; \\ 1 & outdegree(n) = 0 \end{cases}$$

- *Coupling:* To measure coupling we need information about dependencies between system functionalities. By the virtue of its construction, the structure of the FD graph represents the relations between different aggregates. In particular, we compute coupling between two nodes in a slice in terms of their directives. Two directives are said to be coupled if a change in one affects the other. We compute this effect as the probability that such a change occurs and propagates the shortest path ($dist$) between them. Note that the coupling measure is asymmetric. Generalizing, the coupling measure between any two internal nodes $p, q \in V$, where $outdegree(p) > 1, outdegree(q) > 1$ and $D_p \cap D_q = \{\phi\}$ is:

$$Cp(p, q) = \frac{\sum_{d_i \in D_p} \sum_{d_j \in D_q} Cp(d_i, d_j)}{|D_p|.|D_q|}$$

where $Cp(d_i, d_j) = \frac{P(d_j)}{dist(d_i, d_j)}$ and $P(d_j) = \frac{1}{|D_q|}$.

$P(d_j)$ is the probability that directive d_j changes among all other directives associated with the node q.

- *Abstraction Level:* The experimental results of the decomposition algorithm indicate that slices with nodes that exhibit maximum cohesion and decreased coupling are also at higher abstraction levels. We know that abstraction level is related to size; the higher the level of a node, the greater the number of its associated directives. Thus, the decomposition algorithm identifies Capabilities as nodes that exhibit high cohesion and low coupling but are also of increased sizes, which is undesirable from an implementation standpoint.

The decomposition algorithm provides an approach to automate the cohesion and coupling measures. Preliminary experimental results indicate that values computed using these metrics are indicative of desirable software engineering characteristics. In particular, we observe that on an average, in a slice, nodes *viz.* Capabilities that have high cohesion values also exhibit low coupling with other nodes. However, the decomposition approach fails to provide nodes at an abstraction

level that are optimal with respect to size. Therefore, we now explore a reconciliation between the synthesis and decomposition algorithms to determine Capabilities that are optimal with respect to the abstraction levels and the computations of cohesion and coupling.

V. RECONCILIATION

Sections III and IV describe the synthesis and decomposition algorithms to formulate Capabilities. In particular, we observe that the computation of coupling and cohesion values using the *decomposition approach* can be easily automated. This is because the coupling measure is a function of distance of change propagation and probability of change, and therefore, is completely objective. Likewise, the cohesion measure, although less objective, is conveniently computed for all functional abstractions. In contrast, the excessive subjectivity of the synthesis approach presents little scope for automating the formulation of Capabilities in complex emergent systems. However, unlike the decomposition algorithm, the *synthesis approach* provides insights about the optimum abstraction level of a Capability. Hence, we construct a composite algorithm to formulate Capabilities such that it incorporates elements of cohesion and coupling from the decomposition algorithm and that of the abstraction level from the synthesis algorithm. In this section, we first enumerate the steps of the composite algorithm. Then we use the example of the computer based library system to illustrate the reconciliation of the top-down and bottom-up approaches in the composite algorithm.

A. Composite Algorithm

The composite algorithm represents system functionalities at different abstraction levels using an FD graph, as in the decomposition approach. This is because, in the synthesis approach one needs to consider all possible system directives and then determine functional abstractions through an iterative process, which challenges the limited processing capacity of the human mind [21]. In contrast, the decomposition algorithm provides a more structured approach and begins with a single entity — system mission — that is easily comprehensible. Therefore, the input to the composite algorithm is an FD graph representation of the system functionality. The steps of the algorithm are detailed below:

1) Construct an FD graph to represent the system functionality.
2) Determine all possible slices from the FD graph. Each node within a slice is associated with a unique set of directives such that the union of these directive sets is equivalent to entire set of directives of a system.
3) Compute the cohesion and coupling values for nodes in each slice using the metrics defined by the decomposition approach, described in Section IV. Use these values to determine the average cohesion and coupling measures of a slice.
4) Similarly, compute the abstraction levels and sizes of each internal node in a slice.

5) The set of Capabilities is that slice which exhibits high cohesion, low coupling and comprises nodes of balanced abstraction levels.

We now illustrate the steps of the composite algorithm using the example of the library system.

1) Constructing FD Graph: In accordance with the first step of the composite algorithm an FD graph is constructed for the library system. To begin the process of systematic decomposition, we first identify the overall mission: develop software to automate library services. The mission is represented by the root node of the FD graph, which is illustrated in Figure 8. Observe that the mission is partitioned into functionalities of lower abstraction levels. The elliptical vertices in the FD graph denote the directives of the library system and the internal nodes indicate potential Capabilities. Also, the weight associated with an edge symbolizes the relevance value of a directive to its immediate parent node. Figure 8 shows these values as $1, 3, 7,$ or 10. However, they are normalized on a [0,1] scale (as defined in Table I) for the computation of cohesion measure. Note that the weight of an edge between internal nodes is inconsequential and so is denoted by zero.

2) Determining Slices: In our experiment with the library system we computed 1014 valid slices from a possible 1048576 combinations of nodes. In essence there are six basic sets of slices and these are listed in Table II. Permutations of each basic slice set are also valid combinations. For example, for the slice S_1 of Table II, the permutations $\{n_1, n_2, n_3\}$ and $\{n_1, n_3, n_2\}$ are also considered as unique slices. This is primarily because the coupling measure is asymmetric, *i.e.* $Cp(n_i, n_j) \neq Cp(n_j, n_i), i \neq j$; coupling is a function of probability of change which is computed using the size of a node. Consequently, permutations of the basic slice sets also need to be considered as individual slices. We conjecture that the coupling measure can assist in choosing an implementation order of Capabilities that potentially minimizes the impact of change.

TABLE II
Basic Slice Sets

SLICE	NODES	PERMUTATIONS
S_1	n_1, n_2, n_3	2^3
S_2	n_1, n_3, n_4, n_5	2^4
S_3	n_1, n_3, n_4, n_8, n_9	2^5
S_4	n_2, n_3, n_{10}, n_{11}	2^4
S_5	$n_3, n_4, n_5, n_{10}, n_{11}$	2^5
S_6	$n_3, n_4, n_8, n_9, n_{10}, n_{11}$	2^6

An interesting observation from the FD graph of the library system is that nodes n_6 and n_7 are the only nodes that are not a part of any slice. Further analysis reveals the following explanation: Nodes n_6 and n_7 are internal nodes with only directives as their siblings. Moreover, the parents of n_6 and n_7 are internal nodes whose children are directives and internal nodes. The constraint of a slice definition, *viz.* unique membership of directives, disallows n_6 and n_8 or n_7 and n_3 from being a part of the same slice. In addition, the
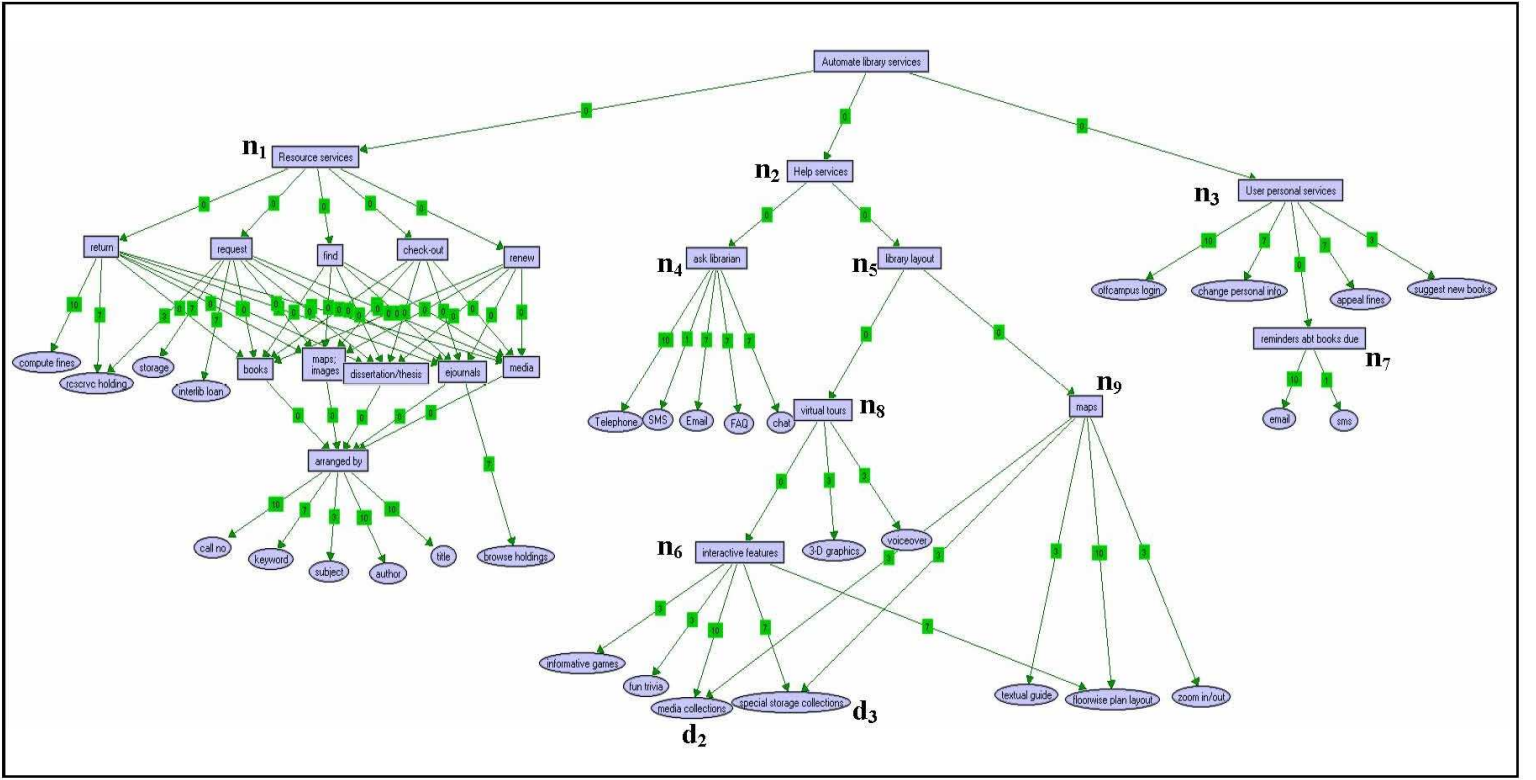
Fig. 8.   *FD Graph of Library System*

requirement for complete coverage of directives of nodes in a slice necessitates the exclusion of n_6 and n_7 from any slice. However, it is possible that n_6 or n_7 can be the root of a large subgraph, in which case their exclusion is detrimental to the formation of Capabilities. This observation compels one to explore the relationship and distribution of internal nodes and directives, when defined at the same level.

3) Computations: Once the slices are determined we compute the cohesion, coupling and abstraction values for each node of a slice. The arithmetic mean of these values provides statistics that help ascertain the quality of a slice from a software engineering perspective. Before discussing the computations, however, we first analyze the relationship between a node's abstraction level, its depth and its size. This is because the objective of the composite algorithm is to formulate Capabilities that not only exhibit high cohesion and low coupling, but are also of balanced abstraction levels. Furthermore, unlike abstraction level, cohesion and coupling are well established concepts that are accepted by the software engineering community. Therefore, it is imperative that we present our notion of an "abstraction level" using insights provided by the synthesis approach and discuss its role in identifying Capabilities.

- *Abstraction Level, Depth and Size:* An abstraction presents information essential to a particular purpose by omitting irrelevant elements. A Capability is a functional aggregate that indicates the functionality expected of the system from a high-level perspective while ignoring minute details. Similarly, the mission of a system is a functional aggregate described at the highest level of ab-

straction because it states the overall system functionality *sans* low-level information. Therefore, with respect to the FD graph, the root is of the highest abstraction and a directive is the lowest. Furthermore, we observe that the abstraction of the internal nodes decreases with their increased distance from the root; distance is the length of the shortest path from the root. We refer to this distance as the *depth* of a node. More specifically, the greater a node's depth, the lower is its abstraction level. Therefore, for a node $n \in V$ in an FD Graph G, the qualitative relation between abstraction level and depth is denoted as:

$$Abstraction(n) \propto \frac{1}{depth(n)}$$

We now discuss the relation between a node's depth and its size. Recall that size is the number of directives associated with a node. The FD graph of the library system indicates that the size of an internal node decreases as its depth increases. For example, in Figure 8 node n_2 has size 14 and depth 1 whereas n_4 is of size 5 and depth 2. We confirm if the random variables, size and depth, are truly correlated by using a scatter plot. Specifically, we plot the average values of size and depth of a node within a slice. This data is obtained from the 1014 slices computed from the FD graph of the library system. The scatter plot is shown in Figure 9 where, within a given slice, the average size of a node is plotted against the average depth of a node. We discuss the following observations about the scatter plot diagram:
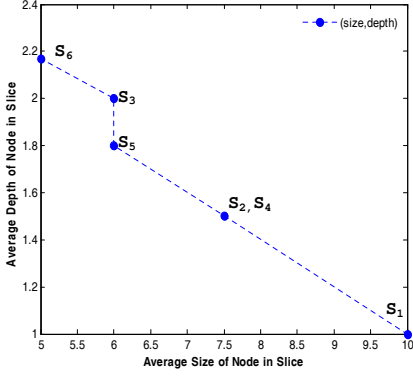
Fig. 9. *Illustrating the relation between depth and size*

library system. Of specific interest are slices that exhibit, on an average, high cohesion and low coupling values. In particular, among all slices computed from the FD graph (Figure 8) of the library system, we examine the slices, PS_1 and PS_2, that have the two highest cohesion values. PS_1 and PS_2 are permutations of the basic slice sets S_1 and S_2 respectively. The cohesion and coupling values of slices PS_1 and PS_2 are presented in Table IV. Note that the cohesion and coupling values of PS_1 and PS_2 are higher and lower respectively, than the median values of the slices of the library system, described in Table III. In particular, slice PS_1 has maximum cohesion and minimum coupling values. Slice PS_2 exhibits the second highest cohesion value and a coupling measure of 1.09577 that is lower than the overall coupling median of 2.76445.

a) Permutations: In the scatter plot of Figure 9, each data point with coordinates $(size, depth)$ corresponds to the basic slice set listed in Table II, and therefore, is denoted by S_i. Although we plot the average size and the depth of a node in each of the 1014 slices computed from the FD graph of the library system, there are only five data points. This is because the permutations of the basic slices results in additional valid slices. However, the average node size and depth values remain unchanged for the permutations of the same basic slice set. Therefore, each point $S_i, (i = 1, \ldots, 6)$ represents the average $(size, depth)$ of a node, which is unchanging in the permutations corresponding to the basic slice sets, shown in Table II.

b) Correlation: The scatter plot shows that there is a relation between the depth and size, which is reaffirmed by the large value of their correlation coefficient $Corr(size, depth) = -0.966630075$. This implies that depth and size are highly negatively correlated. In addition, as discussed earlier, we know that the level of abstraction decreases with an increase in depth. Using this relation and the negative correlation between size and depth, one infers that the size of a node is proportional to its abstraction level. Therefore, for a node $n \in V$ in an FD graph $G = (V, E)$, the qualitative relation between abstraction level and size is denoted as:

$$Abstraction(n) \propto size(n)$$

Hence, the relation between a node's size, depth and abstraction level is used to assist in judicious identification of Capabilities. In particular, we conclude that a balanced abstraction level is influenced by the size of a node.

- *Cohesion and Coupling:* The cohesion and coupling measures are computed using the metrics defined in Section IV. The maximum, minimum, and median average cohesion and coupling values of the slices in the library system are also detailed in Table III, along with the average size and depth values.

4) Selecting Capabilities: We illustrate the final step of the composite algorithm — determining the optimum set of Capabilities from the set of all slices — using the example

According to the decomposition algorithm, slice PS_1 is the most optimal among all slices of the library system. This is because it exhibits maximum cohesion and minimum coupling values when compared to all other slices. In contrast, the synthesis algorithm chooses slice PS_2 as the desirable set of Capabilities. Recall that the synthesis approach emphasizes on low abstraction levels, and consequently, reduced sizes. The average size of each node in PS_1 is 10 where as that of PS_2 is 7.5 as shown in Table V. The implementation size of individual aggregates in PS_1 can be reduced if node n_2 is replaced by n_4 and n_5, which results in a combination of nodes (n_1, n_4, n_5, n_3), *viz.* basic slice set S_2 and in particular, the least coupled permutation, PS_2. This also implies that nodes n_4 and n_5 are at a lower level of abstraction than node n_2. The marginal increase in coupling of PS_2 is offset by the advantage of constructing smaller sized Capabilities. We observe from the FD graph in Figure 8 that there are no intersection edges between nodes n_4 and n_5, signifying that this is a balanced abstraction level. In contrast, let us consider the scenario where we choose to implement nodes, say n_8 and n_9 instead of n_5, which are defined at much lower level of abstraction and are of a smaller size. The node combination $(n_1, n_4, n_8, n_9, n_3)$ is actually the basic slice set S_3 listed in Table II. We choose the permutation $PS_3 = \{n_4, n_9, n_3, n_8, n_1\}$ because it has the lowest coupling among all possible permutations of S_3. Recall that all permu-

tations exhibit the same cohesion, and hence, fails to influence the selection. The cohesion, coupling, size and depth values values are shown in comparison with PS_1 and PS_2 in Table IV and Table V. Slice PS_3 has a lower cohesion and a

TABLE V

Average Size and Depth

Slice	Average Size	Average Depth
PS_1	10	1
PS_2	7.5	1.5
PS_3	6	2

higher coupling value than PS_1 and PS_2. However, it exhibits values that is better than the median cohesion and coupling values of the overall slices of the library system. In addition, PS_3 also has a smaller size when compared to the selections of the decomposition algorithm — PS_1— or the synthesis algorithm — PS_2. Collectively, these factors seem to indicate that this slice is perhaps a more optimal choice than either PS_1 or PS_2 — as illustrated by Table V. However, we observe from the FD graph in Figure 8 that nodes n_8 and n_9 share common directives d_2 and d_3. If these nodes are considered as Capabilities, then directives d_2 and d_3 will be associated with only one Capability. This implies that linkages will be broken to ensure that each directive is bound to either n_8 or n_9. Even so, this does not eliminate the implicit coupling that exists between n_8 and n_9. Consequently, it is difficult to consider nodes n_8 and n_9 as Capabilities with reduced coupling. This nonconformity with a basic tenet of the Capability definition forces us to reject PS_3 as an optimal set, and instead, choose slice PS_2.

To summarize, the composite algorithm constructs an FD graph, determines all possible slices and utilizes the cohesion and coupling computations of the decomposition algorithm to produce potential sets of Capabilities. Then, it incorporates the criteria of a balanced abstraction level and reduced sizes as illustrated by the synthesis algorithm to choose an optimal slice for the library system. Thus, the elements of cohesion and coupling from the top-down approach and the abstraction level from the bottom-up approach constitute the composite algorithm to determine Capability sets for developing change-tolerant systems.

VI. Conclusion

Software engineering methods for system development are often based on a top-down or a bottom-up approach. However, our solution framework, CE, constructs complex emergent systems as change-tolerant entities by utilizing a complementary approach. In particular, we use a composite algorithm to formulate Capabilities as maximally cohesive and minimally coupled functional abstractions of a system. Presently, a Capability depicts only the functionality of a system; integrating non-functional aspects into the definition of a Capability is part of future work. The cohesion and coupling measures of these basic building blocks are computed as in the decomposition algorithm and the abstraction level

as defined by the synthesis algorithm. Note that the former algorithm is based on a top-down approach while the latter, on a bottom-up approach. Thus, the composite algorithm is a blend of the two polar approaches. Experimental results further substantiate the need for such a complementary approach. Our experience in assessing the efficacy of the synthesis and decomposition algorithms aids in understanding the essence of a Capability, and emphasizes that the design of Capabilities is in fact a reconciliation of diametrically opposite approaches to problem solving.

References

[1] F. Heylighen, "Self-organization, emergence and the architecture of complexity," in *Proceedings of the 1st European Conference on System Science*, 1989, pp. 23–32.

[2] D. L. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems," in *Proc. of 7th International Conference on Software Engineering*, Mar. 1984, pp. 408–417.

[3] R. Ravichandar, J. D. Arthur, and S. Bohner, "Capabilities engineering: Constructing change-tolerant systems," in *40th Hawaii International Conference on System Sciences (to appear)*, 2007.

[4] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, Nancy, France, Oct. 1996, pp. 108–124.

[5] S. D. P. Harker, K. D. Eason, and J. E. Dobson, "The change and evolution of requirements as a challenge to the practice of software engineering," in *Proceedings of IEEE International Symposium on Requirements Engineering*, Jan. 1993, pp. 266–272.

[6] Y. Malaiya and J. Denton, "Requirements volatility and defect density," in *10th International Symposium on Software Reliability Engineering*, Los Alamitos, California, Nov. 1999, pp. 285–294.

[7] D. Zowghi and N. Nurmuliani, "A study of the impact of requirements volatility on software project performance," in *9th Asia-Pacific Software Engineering Conference*, Gold Coast, Australia, Dec. 2002, p. 3.

[8] G. A. Tull, "Guide for the preparation and use of performance specifications," Department Of Defense," Headquarters U.S. Army Material Command, 1999.

[9] D. of Defense. (2000) Guidebook for performance-based services acquisition (pbsa) in the department of defense. Acquisition, Technology, and Logistics Web. [Online]. Available: www.acq.osd.mil/dpap/Docs/pbsaguide010201.pdf

[10] M. Montroll and E. McDermott. (2003) Capability-based acquisition in the missile defense agency: Independent research project. Storming Media. [Online]. Available: http://www.stormingmedia.us/82/8242/A824224.html

[11] D. Rowe and J. Leaney, "Evaluating evolvability of computer based systems architectures-an ontological approach," in *International Conference and Workshop on Engineering of Computer-Based Systems*, Mar. 1997, pp. 360–367.

[12] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.

[13] E. Yourdon and L. Constantine, *Structured Design*. Englewood Cliffs, N.J: Prentice-Hall, 1979.

[14] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[15] J. A. Goguen and C. Linde, "Techniques for requirements elicitation," in *Proc. Int. Symp. Req. Engineering*, Los Alamitos, California, 1993, pp. 152–164.

[16] L. B. S. Racoon, "The complexity gap," *ACM SIGSOFT Software Engineering Notes*, vol. 20, pp. 37–44, 1995.

[17] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 1–30, 1996.

[18] S. Lauesen and O. Vinter, "Preventing requirements defects: An experiment in process improvement," *Requirements Engineering*, vol. 6, pp. 37–50, 2001.

[19] J. M. Bieman and L. M. Ott, "Measuring functional cohesion," *IEEE Transactions in Software Engineering*, vol. 20, pp. 644–657, 1994.

[20] B. W. Boehm, *Software Risk Management*. New York: IEEE Computer Society Press, 1989.

[21] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, pp. 81–97, 1956.